Algorithms and Data Structures

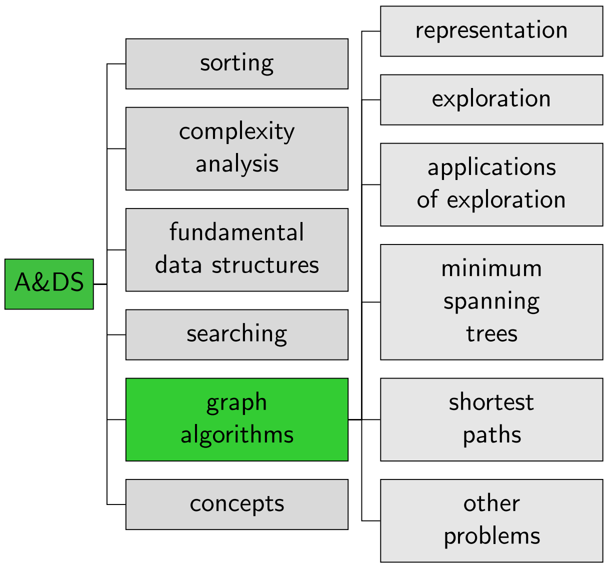
C1. Graphs: Foundations and Exploration

Gabriele Röger and Patrick Schnider

University of Basel

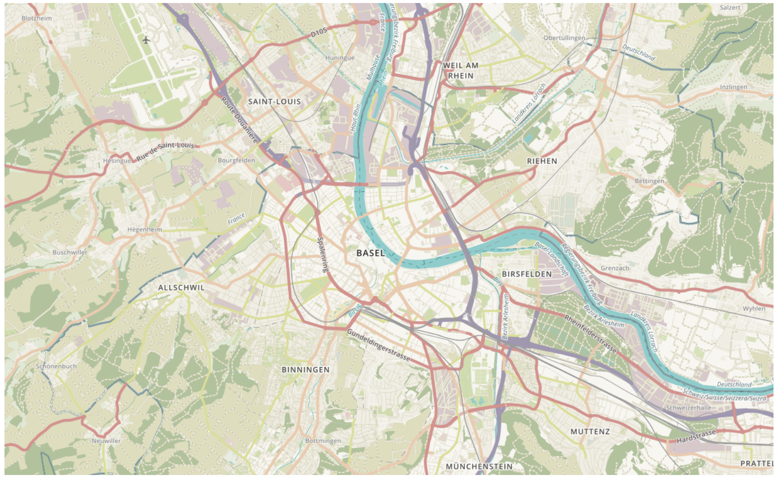
April 24, 2025

Content of the Course



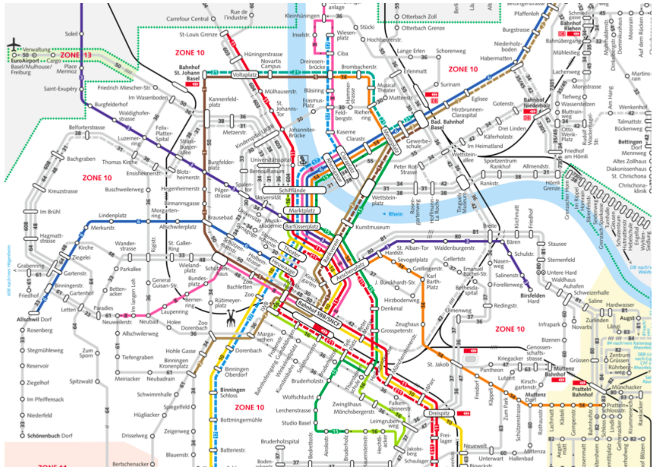
C1.1 Motivation

Street Maps



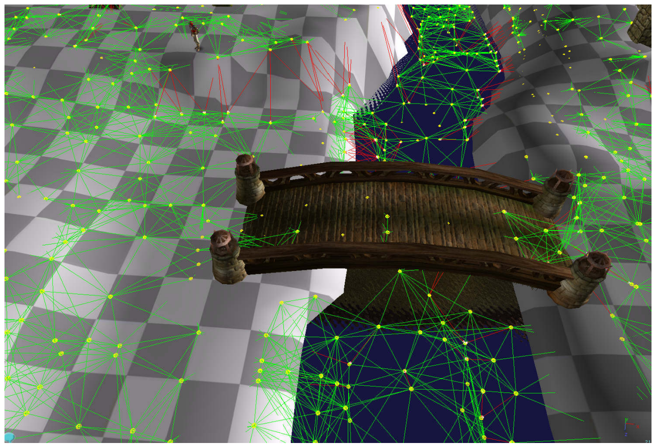
openstreetmap.org

Route Networks



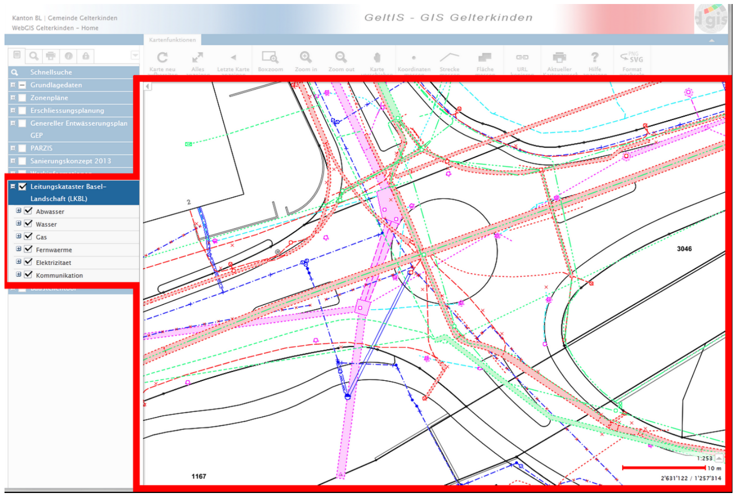
tnw.ch

Navigation Networks in Games



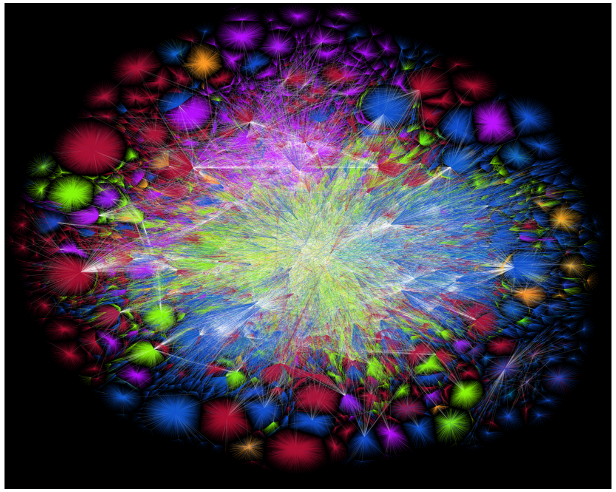
heroengine.com

Urban Supply System



dgis.info

Internet

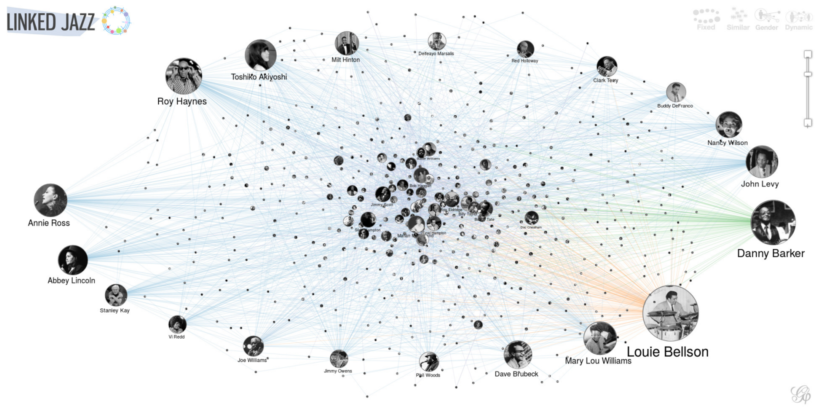


Barrett Lyon / The Opte Project
Visualization of the routing paths of the Internet.

Social Networks

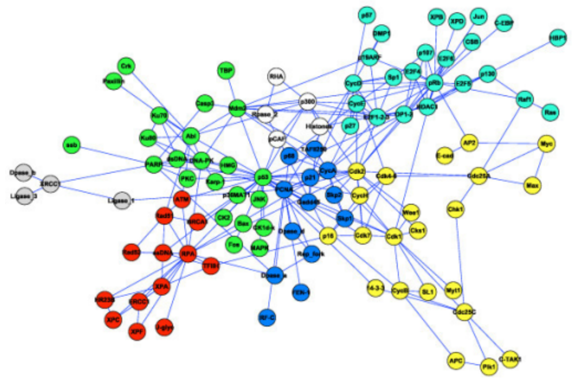
"'Visualizing Friendships"' by Paul Butler

Collaboration



linkedjazz.org

Protein Interaction



Network representation of the p53 protein interactions
Module detection in complex networks using integer optimisation,
Xu G, Bennett L, Papageorgiou LG, Tsoka S - Algorithms Mol Biol (2010)

Possible Questions

- Are A and B connected?
- What is the shortest connection between A and B?
- What is the longest distance between two elements?
- How much water can the sewer system discharge?

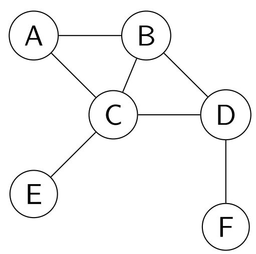
Abstract Graphs

A Graph consists of vertices and edges between vertices.

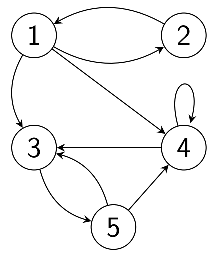
	Vertices	Edges
Streets	Crossing	Street segment
Internet	AS (≈ Provider)	Route
Facebook	Person	Friendship
Proteins	Protein	Interaction

C1.2 Definition

Undirected and Directed Graphs



undirected graph



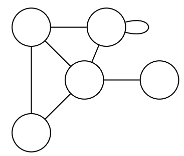
directed graph

Graphs

- A graph is a pair (V, E) comprising
  - V: finite set of vertices
  - E: finite set of edges
- Every edge connects two vertices u and v
  - undirected graph: set $\{u, v\}$
  - directed graph: pair (u, v)
- Multigraphs permit multiple parallel edges between the same nodes.
- Weighted graphs associate each edge with a weight (a number).

Undirected Graphs: Terminology

- Neighbours of a vertex u: all vertices v with $\{u, v\} \in E$.
- degree(v): Degree of a vertex = Number of neighbours.
  - Exception: Self-loops increase the degree by 2.
    Self-loop = edge that connects a vertex with itself.

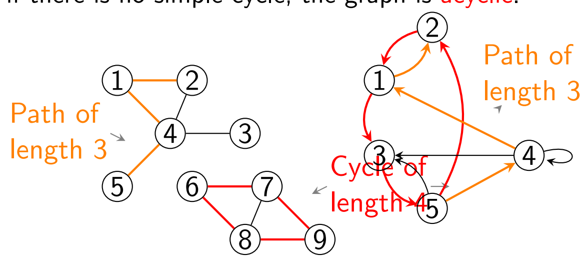


Directed Graphs: Terminology

- Successors of vertex u: all vertices v with $(u, v) \in E$.
- Predecessors of vertex u: all vertices v with $(v, u) \in E$.
- outdegree(v): outdegree = number of successors
- indegree(v): indegree = number of predecessors

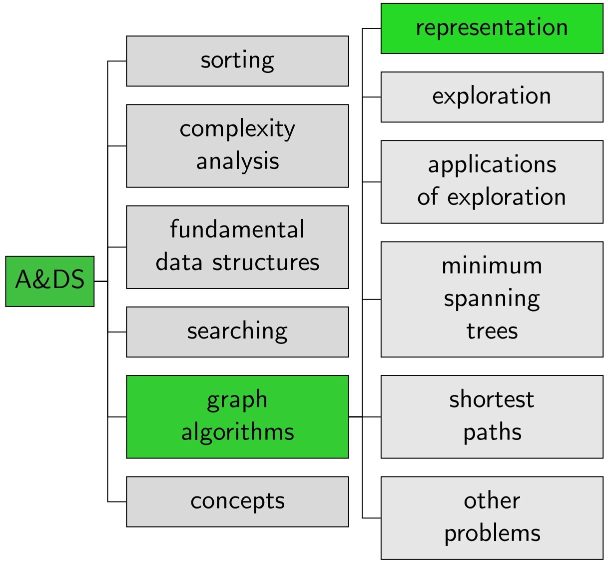
Paths and Cycles

- Path of length n: Sequence $(v_0, \ldots, v_n)$ of vertices with
  - $\{v_i, v_{i+1}\} \in E$ for $i = 0, \ldots, n-1$ (undirected graph)
  - $(v_i, v_{i+1}) \in E$ for $i = 0, \ldots, n-1$ (directed graph)
  - The path is simple if all vertices are distinct.
  - Example: (5,4,1,2)
- Cycle: Path with equal start and end vertex ($v_0 = v_n$) of length > 0.
  - (6,7,9,8,6) in the undirected and (5,2,1,3,5) in the directed example graph
  - The cycle is simple if all vertices $v_1, \ldots, v_n$ are distinct.
  - if there is no simple cycle, the graph is acyclic.



C1.3 Representation

Content of the Course



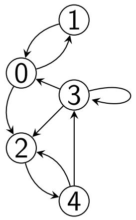
Representation of Vertices

- We use numbers 0 to $|V| - 1$ for the vertices.
- If not the case in application: Us a map to convert from names to numbers.

Adjacency-matrix Representation

Graph $G = (\{0, \dots, |V| - 1\}, E)$ represented as $|V| \times |V|$ matrix with entries a_{ik} (in row i, column k):

$$a_{ik} = \begin{cases} 1 & \text{if } (i,k) \in E \text{ (directed graph) or} \\ & \{i,k\} \in E \text{ (undirected graph)} \\ 0 & \text{otherwise} \end{cases}$$

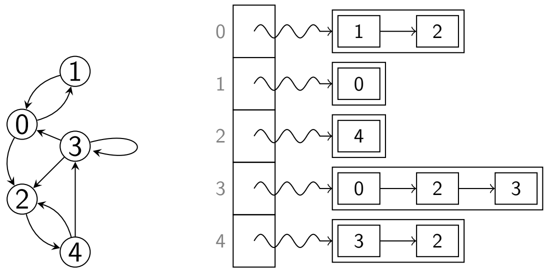


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

For undirected graphs symmetric

Adjacency-list Representation

Store for every vertex the list of successors / neighbours.



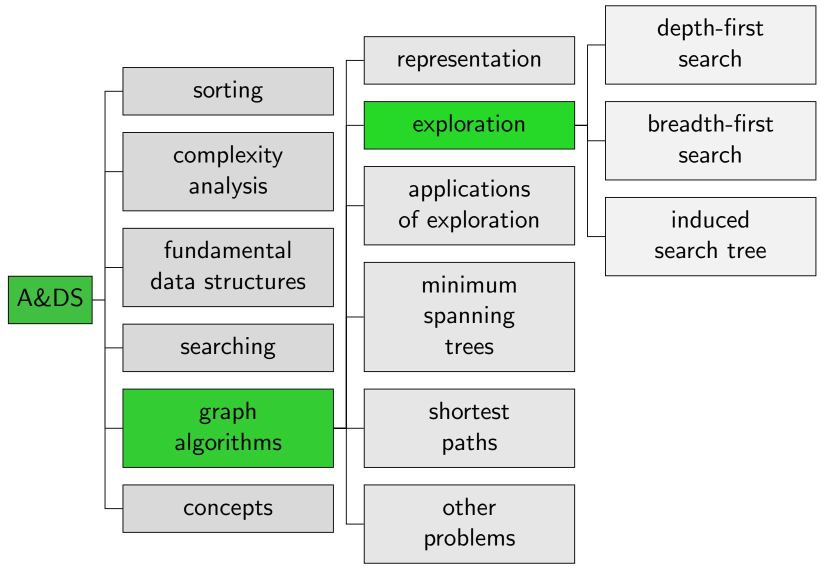
Representation: Complexity

	Adj. matrix	Adj. list
Space	$\lvert V\rvert^2$	$\lvert E\rvert + \lvert V\rvert$
Add edge	1	1
Edge between u and v?	1	(out)degree(v)
Iterate over outgoing edges	$\lvert V\rvert$	(out)degree(v)

Often sparse graphs (low average degree)
Which representation?

C1.4 Graph Exploration

Content of the Course



Graph Exploration

- Task: Given a vertex v, visit all vertices that are reachable from v.
- Often used as ingredient of other graph algorithms.
- Depth-first search: go "deep" into the graph (away from v)
- Breadth-first search: first all neighbours, then neighbours of neighbours, ...

Depth-first Search

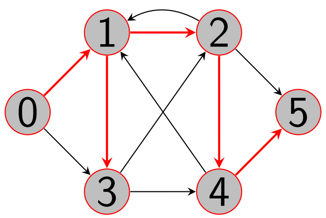
Mark visited vertices

- Mark v
- Iterate over the successors/neighbours w of v.
  - If w not marked, start recursively from w.

Abbreviation: DFS

Depth-first Search: Example

Here: Visit successors in increasing order of their number.



Depth-first search from start vertex 0 marks vertices in order 0 - 1 - 2 - 4 - 5 - 3

Depth-first Search: Algorithm (Recursive)

```python
def depth_first_exploration(graph, node, visited=None):
    if visited is None:
        visited = set()
    if node in visited:
        return
    visited.add(node)
    for s in graph.successors(node):
        depth_first_exploration(graph, s, visited)
```

If we expect that most vertices will be visited: bool array instead of set for `visited`

Depth-first Vertex Orders

- Preorder: Vertex is included before its children are considered.
- Postorder: Vertex is included when the (recursive) depth-first search of all its children has finished.
- Reverse Postorder: Like postorder, but in reverse order.

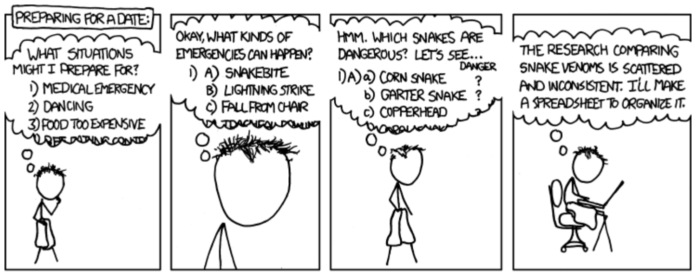
```
def depth_first_exploration(graph, node, visited):
    if node in visited:
        return
    preorder.append(node)
    visited.add(node)
    for s in graph.successors(node):
        depth_first_exploration(graph, s, visited)
    postorder.append(node)
    reverse_postorder.appendleft(node)
```

(Representation of vertex sequence as a deque.)

Depth-first Search: Algorithm (Iterative)

```
def depth_first_exploration(graph, node):
    visited = set()
    stack = deque()
    stack.append(node)
    while stack:
        v = stack.pop()  # LIFO
        if v not in visited:
            visited.add(v)
            for s in graph.successors(v):
                stack.append(s)
```


Depth-first Search in Practice



https://xkcd.com/761/

Breadth-first Search

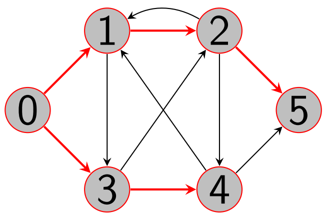
First all neighbours, then neighbours of neighbours, ...

- Mark v
  $\rightarrow$ Distance 0
- Mark all unmarked successors/neighbours of v
  $\rightarrow$ Distance 1
- Mark all unmarked successors/neighbours of vertices with distance 1.
- Mark all unmarked successors/neighbours of vertices with distance 2.
- ...
- Until vertices of distance i do not have unmarked successors/neighbours.

Abbreviation: BFS

Breadth-first Search: Example

Here: Visit successors in increasing order of their number.



Breadth-first search from start vertex 0 marks vertices in order 0 - 1 - 3 - 2 - 4 - 5

Breadth-first Search: Algorithm (Conceptually)

Only difference to iterative depth-first search:
First-in-first-out treatment of vertices (instead of last-in-first-out)

```
def breadth_first_exploration(graph, node):
    visited = set()
    queue = deque()
    queue.append(node)
    while queue:
        v = queue.popleft()  # FIFO
        if v not in visited:
            visited.add(v)
            for s in graph.successors(v):
                queue.append(s)
```


Breadth-first Search: Algorithm (Somewhat more Efficient)

We only further consider a vertex when we first run across it.
We can directly mark it as visited and disregard it if we see it again.

```
def breadth_first_exploration(graph, node):
    visited = set()
    queue = deque()
    visited.add(node)
    queue.append(node)
    while queue:
        v = queue.popleft()
        for s in graph.successors(v):
            if s not in visited:
                visited.add(s)
                queue.append(s)
```

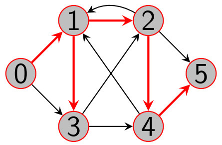

Running Time

For all algorithm variants:

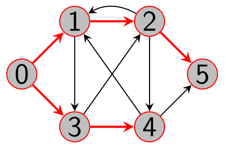
- Every reachable vertex gets marked.
- We follow every reachable edge exactly once.
- Running time $O(|V| + |E|)$
  - We can restrict this to the reachable vertices and edges.

Induced Search Tree

The induced search tree of a graph exploration contains for every visited vertex (except for the start vertex) an edge from its predecessor in the exploration.



depth-first search



breadth-first search

(induced search tree $\neq$ binary search tree)

Induced Search Tree: Example BFS

- Every vertex has at most one predecessor in the tree.
- Represent induced search tree by the predecessor relation.
- The visited vertices are exactly those for which there is a predecessor set: We do not need `visited` anymore.

```
def bfs_with_predecessors(graph, node):
    predecessor = [None] * graph.no_nodes()
    queue = deque()
    # use self-loop for start node
    predecessor[node] = node
    queue.append(node)
    while queue:
        v = queue.popleft()
        for s in graph.successors(v):
            if predecessor[s] is None:
                predecessor[s] = v
                queue.append(s)
```


C1.5 Summary

- Graphs consist of vertices and edges.
- Edges can be directed or undirected.
- Graph exploration systematically visits all vertices that can be reached from the given vertex.
  - Depth-first search goes “deeper” into the graph whenever possible.
  - Breadth-first search first visits the vertices that are closer to the start vertex.